Improved Utilization and Responsiveness with Gang Scheduling

Dror G. Feitelson
Institute of Computer Science
The Hebrew University of Jerusalem
91904 Jerusalem, Israel

Morris A. Jette
Livermore Computing
Lawrence Livermore National Laboratory
Livermore, CA 94550

Abstract

Most commercial multicomputers use space-slicing schemes in which each scheduling decision has an unknown impact on the future: should a job be scheduled, risking that it will block other larger jobs later, or should the processors be left idle for now in anticipation of future arrivals? This dilemma is solved by using gang scheduling, because then the impact of each decision is limited to its time slice, and future arrivals can be accommodated in other time slices. This added flexibility is shown to improve overall system utilization and responsiveness. Empirical evidence from using gang scheduling on a Cray T3D installed at Lawrence Livermore National Lab corroborates these results, and shows conclusively that gang scheduling can be very effective with current technology.

1 Introduction

As parallel computers become more popular, there is a growing need for good schedulers that will manage these expensive shared resources. And indeed, many scheduling schemes have been designed, evaluated, and implemented in recent years [5, 10].

Many papers investigate scheduling schemes from a system point of view, asking what the system can do to improve utilization and response time, but disregarding the effect on the user. As a result they sometimes advocate solutions that require users to depart from common practice, e.g. to write applications in a style that supports dynamic partitioning (i.e. the allocation may change at runtime) [30, 20], rather than the prevalent SPMD style.

We take a different approach, and ask what the system can do given the constraint that users require jobs to execute on a fixed number of processors (as in SPMD). Within this framework, we compare variable partitioning, possibly with reordering of the jobs in the queue, with gang scheduling. We show that although gang scheduling suffers from more overhead than variable partitioning, it can lead to significant improvements due to its added flexibility. Indeed, gang scheduling can actually give better service (reduced response time) *and* improved utilization, so using it leads to a win-win situation relative to variable partitioning.

The results agree with actual experience on the LLNL Cray T3D, which employs a home-grown gang scheduler [12, 17] (the original system software uses variable partitioning). When this scheduler was ported to the new Cray machine, utilization nearly doubled from 33.4% to 60.9% on average. Additional tuning has led to weekly utilizations that top 96%.

2 Approaches to Scheduling Jobs of Given Size

The schedulers of most commercial parallel systems use variable partitioning. The user specifies the number of processors to use at the time of submitting the job. The scheduler than carves out a partition of the required size, and dedicates it to the job for the duration of its execution. If the required number of processors is not available, the job is either rejected or queued. In most systems a time limit is also imposed, and if the job exceeds it it is killed.

The problem with this scheme is that scheduling decisions have a potentially large, persistent, and unpredictable impact on the future. Specifically, when a

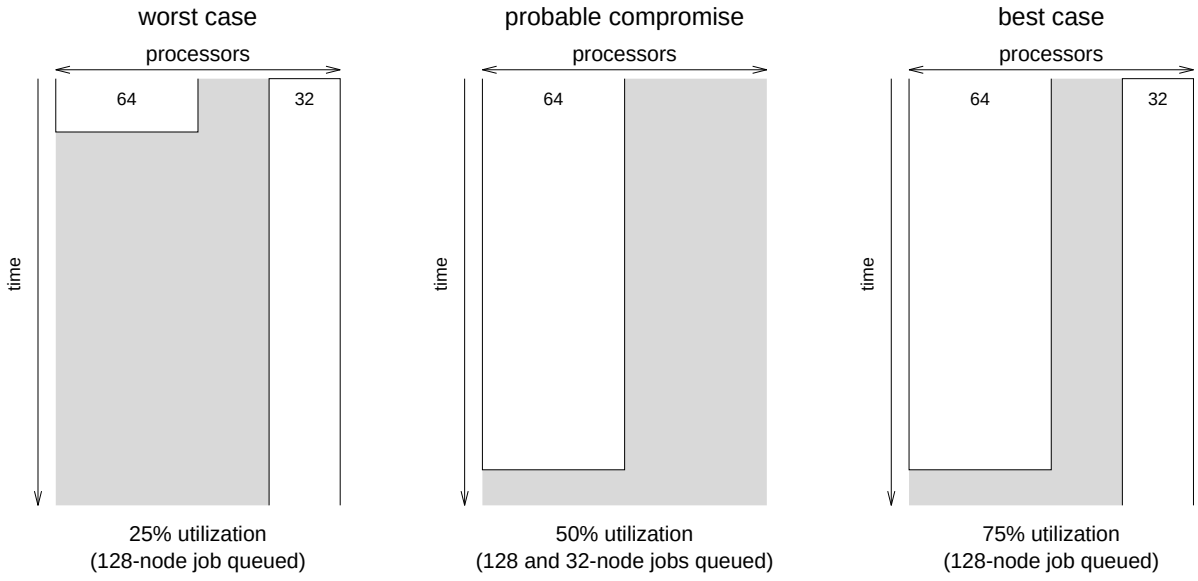

Figure 1: *Example of the problems faced by variable partitioning.*

new job arrives, the system is faced with the following dilemma:

- if the new job can be accommodated, then scheduling it immediately will utilize unused resources, so it is good.
- however, if this job runs for a long time, and will block other jobs in the future, it may lead to more future loss than current gain. So maybe it should be left aside.

Consider the following simple case as an example (Fig. 1): a 128-node system is currently running a 64-node job, and there are a 32-node job and a 128-node job in the queue. The question is, should the 32-node job be scheduled to run concurrently with the 64-node job? Two outcomes are possible. If the 32-node job is scheduled and it terminates before the 64-node job, resource utilization is improved from 50% possibly up to 75%. But if the 64-node job terminates soon after the 32-node job is scheduled, and the 32-node job runs for a long time, the utilization drops from 50% to 25%. And, in order not to starve the 128-node job, it might be necessary to just let the 64-node job run to completion, and settle for 50% utilization.

As the future is usually unknown, there is no solution to this dilemma, and any decision may lead to fragmentation. Thus using variable partitioning may lead to significant loss of computing power [18, 33], either because jobs do not fit together, or because processors are intentionally left idle in anticipation of future arrivals [26].

The most common solution is to reorder the jobs in the queue so as to pack them more tightly [16]. One promising approach is to allow small jobs to move forward in the queue if they can be scheduled immediately. However, this may cause starvation of large jobs, so it is typically combined with allowing large jobs to make reservations of processors for some future time. Only *short* jobs are then allowed to move ahead in the queue (Fig. 2) [3, 19].

The problem with this idea is that it requires information about job runtimes. A rough approximation may be obtained from the queue time limit (in most systems users may choose which queue to use, the difference being that each queue has a distinct set of resource limits associated with it). The idea is that the user would choose the queue that best represents the application's needs, and the system would then be able to select jobs from the different queues to create a job mix that uses the system's resources effectively [31]. However, experience indicates that this information is unreliable, as shown by the distributions of queue-time utilization in Fig. 3. The graphs show that users tend to be extremely sloppy in selecting the queue, thus undermining the whole scheme. (The graphs show the distributions in buckets of 4 percentage points. Thus the top left data point in

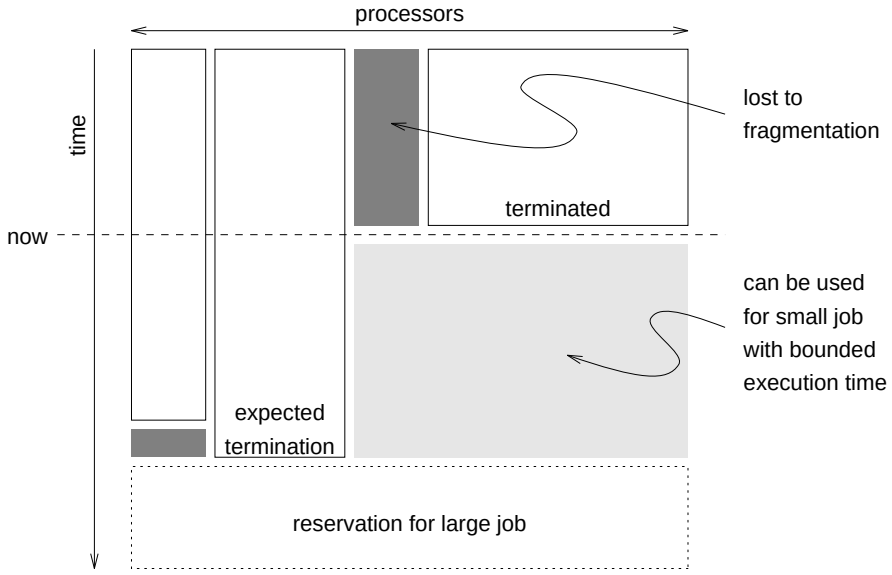

Figure 2: *Runtime bounds on executing jobs allow reservations to be made for large jobs and then backfilling with smaller jobs to reduce fragmentation.*

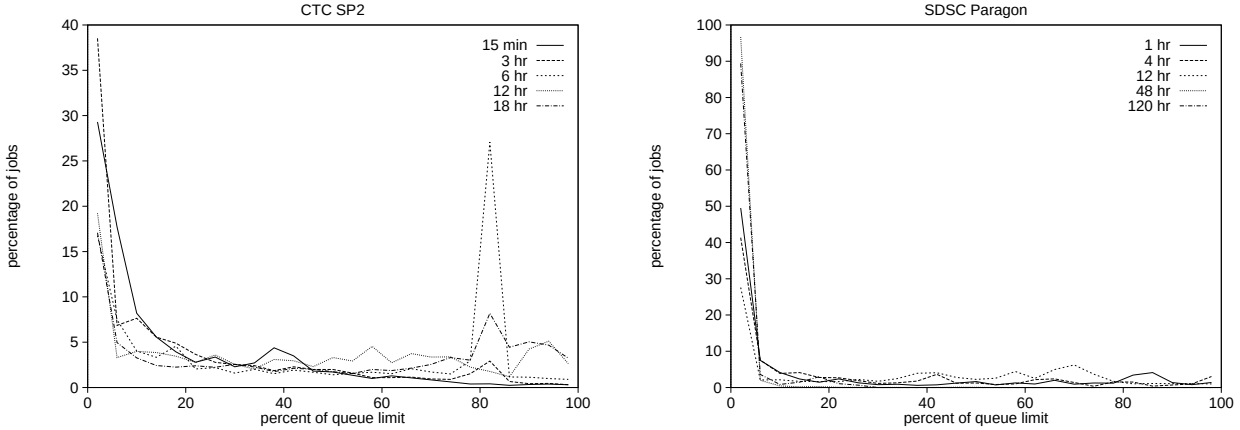

Figure 3: *Job runtimes as a fraction of the batch queue time limit, showing that most jobs use only a fraction of the time limit, even for queues with very long limits. The plot for each queue limit is normalized independently.*

the left graph shows that about 38% of the jobs submitted to all the 3-hour queues on the Cornell SP2 only used between 0 and 4% of their time limit, i.e. they were actually shorter than 7 minutes.)

Another solution to the fragmentation problem is to use adaptive partitioning rather than variable partitioning [29, 28, 25]. The idea here is that the number of processors used is a compromise between the user's request and what the system can provide. Thus the system can take the current load into account, and reduce the partition sizes under high load conditions. However, this scheme also requires a change of user interfaces, albeit much less disruptive than dynamic partitioning.

The prefered solution is to use gang scheduling [22, 7, 8, 27]. With gang scheduling, jobs receive the number of processors requested, but only for a limited time quantum. Then a "multi-context-switch" is performed on all the processors at once, and another job (or set of jobs) is scheduled instead. Thus all

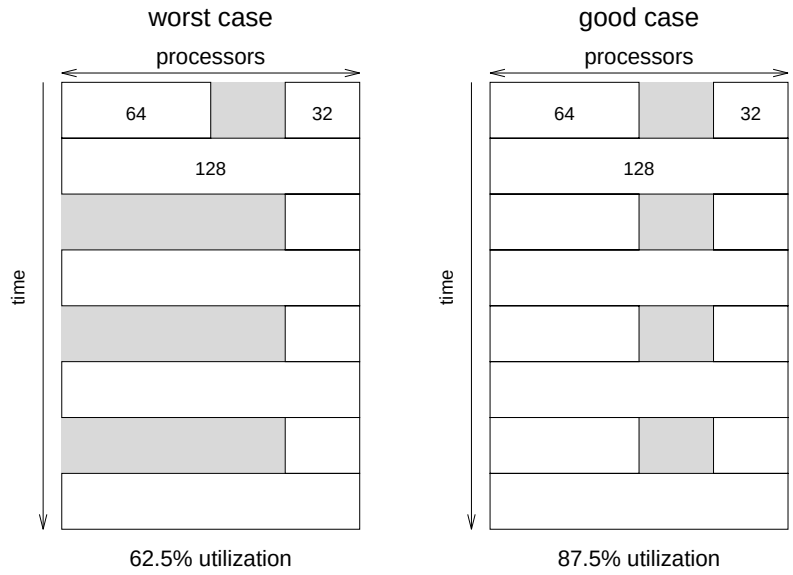

Figure 4: *Example of how the flexibility afforded by time slicing can increase system utilization; compare with Fig. 1.*

jobs can execute concurrently using time slicing, as in conventional uniprocessors. As a result, a scheduling decision only impacts the scheduling slot to which it pertains; other slots are available to handle other jobs and future arrivals. this adds flexibility and boosts performance.

Returning to the example considered earlier, the situation with gang scheduling is illustrated in Fig. 4. The 32-node job can safely run in the same time-slot with the 64-node job, while the 128-node job gets a separate time-slot. There is no danger of starvation. As long as all three jobs are active, the utilization is 87.5%. Even if the 64-node job terminates, leaving the 32-node job to run alone in its time-slot, the utilization is 62.5%. Naturally, a few percentage points should be shaved off these figures to account for context-switching overhead. Nevertheless, this is a unique case where time-slicing, despite its added overhead, can lead to better resource utilization than batch scheduling.

Using gang scheduling not only improves utilization — it also reduces mean response time. It is well known that mean response time is reduced by the shortest-job-first discipline. In workloads with high variability this is approximated by time slicing, because chances are that a new job will have a short runtime [24, 23]. As production workloads do indeed exhibit a high variability [6], it follows that gang scheduling will reduce mean response time. Indeed, gang scheduling has even been advocated in conjunction with dynamic partitioning [21].

3 Simulation Results

3.1 The Compared Scheduling Schemes

In order to demonstrate the ideas described above, we simulate the performance of a multicomputer subjected to a realistic workload and using one of a set of different scheduling schemes. these are:

FCFS: the base case we use for comparison is variable partitioning with first-come-first-serve queuing. This scheme is expected to suffer from significant fragmentation.

Backfill: backfilling was developed for the Argonne National Lab SP1 machine [19], and has recently also been installed on the Cornell SP2 and other machines. It allows short jobs to move forward in the queue provided they do not cause delays for any other job. Only jobs that do not cause delay are moved forward. We assume the scheduler has perfect information when making such decisions, i.e. it knows the exact runtimes of all the jobs in the queue.

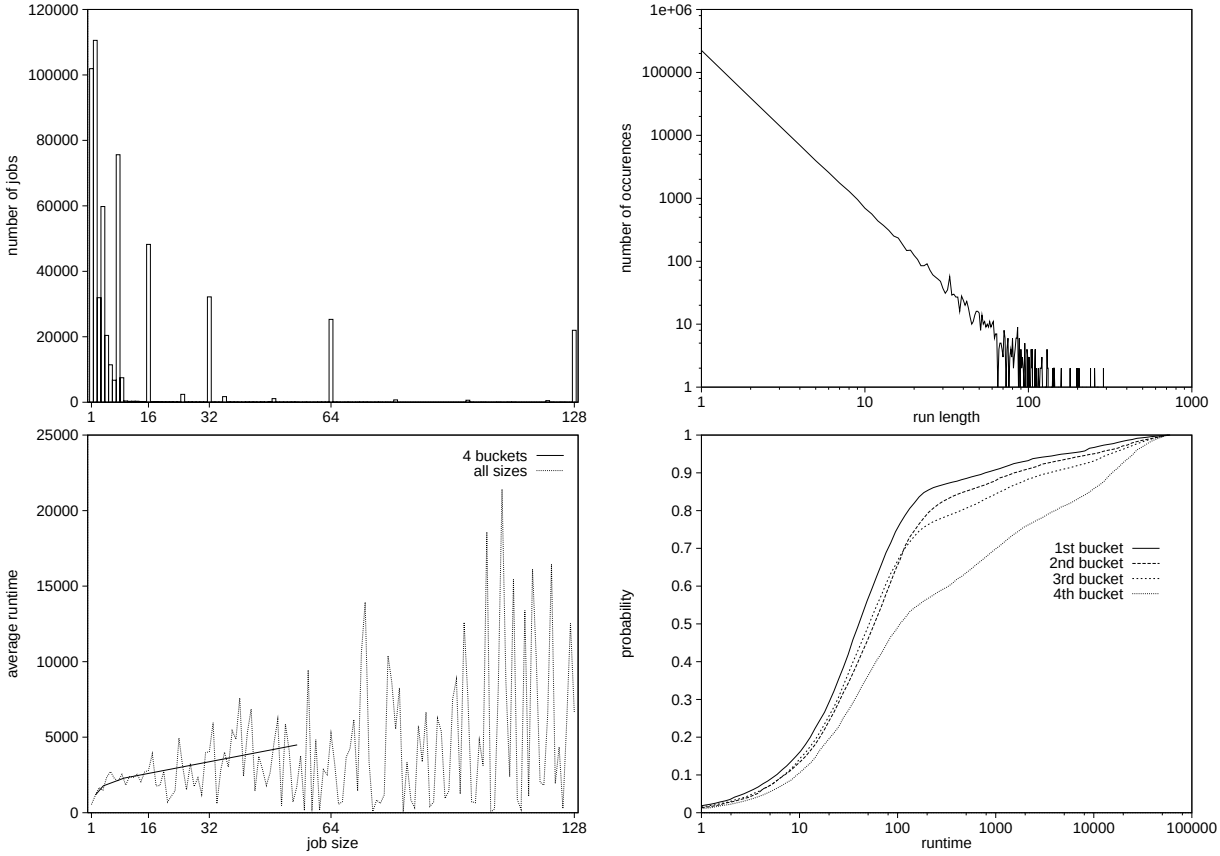

Figure 5: *Statistical properties of the workload model: (a) distribution of job sizes (b) distribution of runlengths (number of repeated executions) (c) correlation of runtime with job size, for all sizes and when jobs are grouped into four buckets according to size (d) cumulative distributions of runtimes for the jobs in the four buckets.*

Prime: this policy is a simplified version of a policy used on the SP2 machine at NASA Ames [14]. The idea is to distinguish between prime time and non-prime time[1]: during prime time, large jobs (more than 32 nodes) are restricted to 10 minutes, while small jobs are allowed up to 4 hours provided at least 32 nodes are available. Thus, if only a few nodes are available, all jobs are restricted to 10 minutes, and responsiveness for short jobs is improved. This achieves a similar effect to setting aside a pool of nodes for interactive jobs [31]. During non-prime time these restrictions are removed. Again, we assume the scheduler knows the runtimes of all jobs.

Gang: gang scheduling with no information regarding runtimes. The jobs are packed into slots using the buddy scheme, including alternate scheduling [4]. Two versions with different scheduling time quanta are compared: one has relatively small time quantum of 10 seconds, so most jobs effectively run immediately, and the other has a time quantum of 10 minutes (600 seconds), so jobs may be queued for a certain time before getting to run.

[1] Our workload model does not include a daily cycle of job submittals — it is a continuous stream of jobs with the same statistical properties. Thus in our simulations the distinction is only in the scheduling policy, which is switched every 12 hours.

3.2 Simulation Methodology

The workload model is an improved version of the model used in [4]. It is based on workload analysis

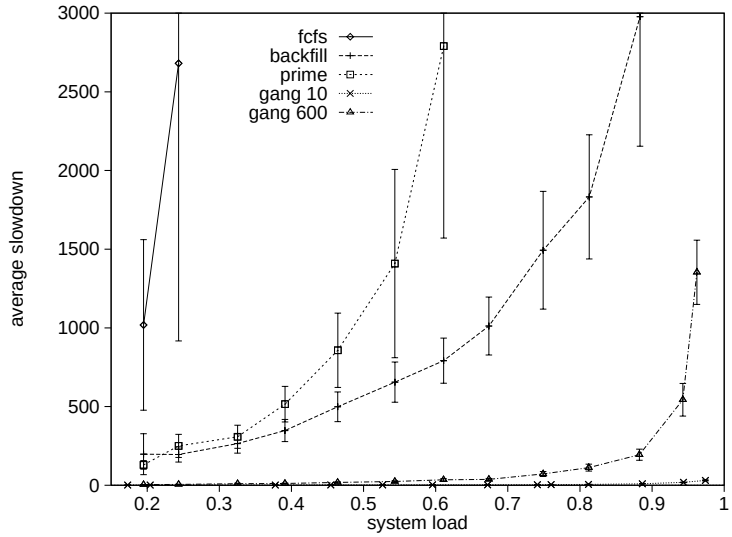

Figure 6: *Simulation results.*

from a number of production systems [6, 15, 32], and is characterized as follows (Fig. 5):

- The distribution of job sizes emphasizes small jobs and powers of two.
- The distribution of runtimes is a three-stage hyperexponential, where the relative weights of the three stages depend on the job size. This dependence is used to create a correlation between the job size and the runtime.
- The arrivals are Poisson, except for jobs that are re-run a number of times, in which case they are re-submitted immediately upon completion.

The simulation uses the batch means method to evaluate confidence intervals. Each batch includes 3333 job terminations. The first batch was discarded to account for simulation warmup. The length of each experiment (i.e. the simulation for each data point in the results) is at least 3 batches, or more as required so that the 90% confidence interval is no larger than 10% of the data point value, up to a maximum of 100 batches. Interestingly, simulations of all scheduling schemes except gang scheduling with short time quanta used all 100 batches, without a significant reductions in the confidence interval: it was typically in the range of 20-40% of the data point value. This reflects the very high variance present in the workload.

The sequence of job arrivals is generated once and reused for each data point and each scheme. Only the mean interarrival time is changed to create different load conditions.

The performance metric is the average slowdown. The slowdown for a given job is defined as the ratio between its response time on a loaded system (i.e. its queuing time plus run time and possible preempted time) and its runtime on a dedicated system.

3.3 Experimental Results

The results are shown in Fig. 6. As expected, FCFS saturates at extremely low loads, and even before saturation it tends to create very high slowdowns. Backfilling and delaying large jobs to non-prime time are both much better, but backfilling can sustain a higher load and produces lower slowdowns. Attempts to improve the performance of the prime/non-prime policy by fiddling with its parameters (the threshold between small and large jobs, and the length of the prime shift) showed that it is relatively insensitive to the exact values of these parameters. However, it should be remembered that our workload model is not suitable for a detailed study of the prime/non-prime policy, because it does not include a daily cycle.

Gang scheduling, even with relatively long quanta of 10 minutes, takes the cake: the slowdowns are very low, and saturation is delayed until system load approaches 1. This agrees with the informal arguments

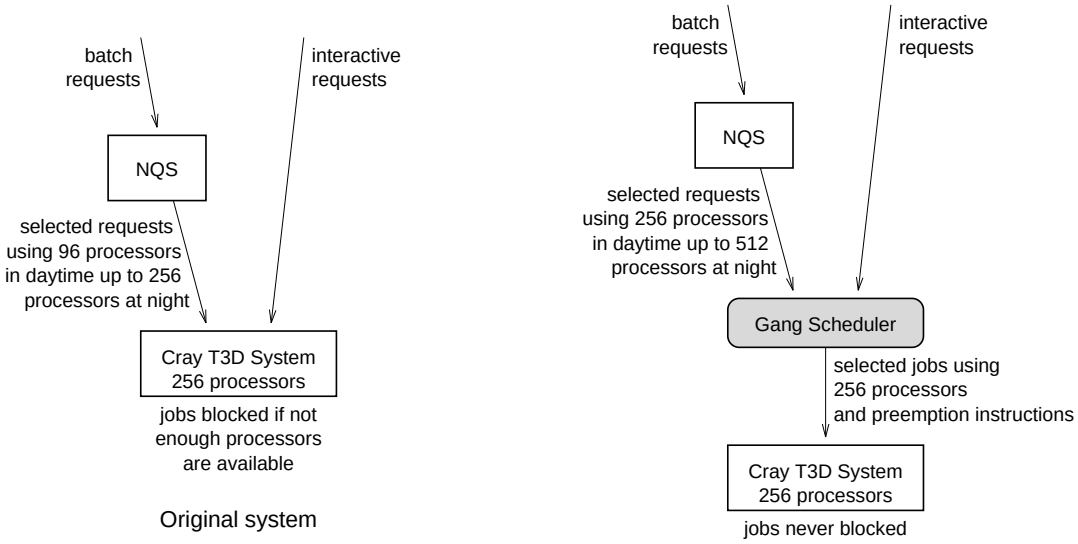

Figure 7: *The place of the gang scheduler in the Cray T3D scheduling system.*

presented in Section 2. While our simulations may be criticized for not modeling the overheads involved in gang scheduling — for example, the overhead of context switching and the effect of corrupting cache state — we feel that with long enough time quanta these overheads can be kept relatively low, so the main results remain valid.

4 Experience with Gang Scheduling on the Cray T3D

The default mode of operation for the Cray T3D is that of variable partitioning. Generally, jobs are allocated a partition of processors as soon as a suitable set becomes available. In the case where a job has waited for longer than some configurable period of time, the initiation of all other jobs is deferred until it begins execution. The partitions are held until the job completes and relinquishes them, effectively locking out any other use for those processors. With such a processor allocation mechanism, the computational requirements of long-running production jobs directly conflict with those of interactive code development work.

Our goals in the development of the Gang Scheduler for the Cray T3D were:

- Provide better response for interactive jobs that are submitted directly
- Provide better throughput for batch jobs that are submitted through NQS
- Permit larger jobs to be executed
- Provide optimum throughput for specific jobs, as designated by management

While achieving all of these objectives would seem impossible, our initial utilization rate of 33.4% provided us a great deal of room for improvement.

In a nutshell, our approach is as follows (Fig. 7). Originally, batch jobs were submitted via NQS and interactive jobs were submitted directly to the Cray system software. NQS buffered the jobs, and forwarded only few small jobs during the day, so that (hopefully) sufficient processors would be left free for interactive use. At night, NQS would submit enough jobs to fill the whole machine. With the gang scheduler, NQS fills the machine during the day and over-subscribes it during the night. The gang scheduler preempts jobs as necessary in order to provide timely service to higher-priority jobs, notably interactive jobs. This is a "lazy" form of gang scheduling: rather than performing a context switch across the whole machine at given intervals, specific jobs are chosen for preemption when an urgent need for processors arises.

Gang scheduling thus effectively creates a larger virtual machine, and meets the above objectives by:

- Time sharing processors for interactive jobs

when experiencing an extremely heavy interactive workload
- Keeping processors fully utilized with batch jobs until preempted by an interactive or other high priority job
- Making processors available in a timely fashion for large jobs
- Making processors available in a timely fashion for specific jobs and making those jobs nonpreemptable

Some might argue that interactive computing with a massively parallel computer is unreasonable, but interactive computing accounts for a substantial portion of our workload and is used for code development and rapid throughput. Interactive jobs currently account for 79% of all jobs executed and 11% of all CPU cycles used in our environment. A single interactive job can be allocated up to 25% of all processors and memory. The aggregate of all interactive work will normally consume between zero and 150% of all processors and memory in our environment. While the Cray T3D is well suited for addressing the execution of grand challenge problems, we wanted to expand its range of functionality into general purpose support of interactive work as well.

4.1 Cray T3D Design Issues

The Cray T3D is a massively parallel computer incorporating DEC alpha 21064 microprocessors, capable of 150 MFLOPS peak performance. Each processor has its own local memory. The system is configured into nodes, consisting of two processors with their local memory and a network interconnect. The nodes are connected by a bidirectional three-dimensional torus communications network. There are also four synchronization circuits (barrier wires) connected to all processors in a tree shaped structure. The system at Lawrence Livermore National Laboratory (LLNL) has 256 processors, each with 64 megabytes of DRAM. Disk storage is required to store the job state information for preempted jobs. This can be either shared or private storage space. We have created a shared 48 gigabyte file system for this purpose. The storage requirements will depend upon the T3D configuration and the amount of resource oversubscription permitted.

Without getting into great detail, the T3D severely constrains processor and barrier wire assignments to jobs. Jobs must be allocated a processor count which is a power of two, with a minimum of two processors (one node). The processors allocated to a job must have a specific shape with specific dimensions for a given problem size. For example, an allocation of 32 processors must be made with a contiguous block with 8 processors in the X direction, 2 processors in the Y direction and 2 processors in the Z direction. Furthermore, the possible locations of the processors assignments is restricted. These very specific shapes and locations for processor assignments are the result of the barrier wire structure. Jobs must be allocated one of the four barrier wires when initiated. The barrier wire assigned to a job cannot change if the job is relocated and, under some circumstances, two jobs sharing a single barrier wire may not be located adjacent to each other. The number of processors assigned to a job can not change during execution [2].

There are two fundamentally different ways of providing for timesharing of processors. The entire state of a job, including memory contents, register contents and switch state information can be written to disk. Alternately, the register and switch state information can be saved and the memory shared through paging. Saving the entire job state clearly makes context switches very time consuming, however, it can provide a means of relocating jobs to different processors and provide a means of preserving executing jobs over computer restarts. Sharing memory through paging can make for much faster context switches. Our system provides timesharing by saving the entire state of a job to disk. Cray does not support paging on this architecture because of the large working sets typical of programs executed and in order to reduce system complexity.

Timesharing by saving the entire state of a job to disk has an additional advantage as well. Given the T3D's constraints on processor assignment, the ability to relocate jobs with this mechanism clearly make it preferable. While the ability to preserve executing jobs over computer restarts has proven to be of some use, most programs complete in a few hours and can be restarted without substantial impact upon the system. Unfortunately, the high context switch time provides lower interactivity than would be desirable. It should also be noted that the system only supports the movement of a job's state in it's entirety. It is not possible to initiate state transfers on a processor by processor basis, although that capability would improve the context switch time.

Job Class	*Priority*	*Wait Time*	*Do-not-disturb Time Multiplier*	*Processor Limit*
Interactive	4	0 Sec	10 Sec	256
Debug	4	300 Sec	1 Year	96
Production	3	1 Hour	10 Sec	256
Benchmark	2	1 Year	1 Year	64
Standby	1	1 Year	3 Sec	256

Table 1: *Scheduling parameters for different job classes.*

The original version of this Gang Scheduler was developed for the BBN TC2000 computer. The BBN computer permitted programs to be assigned processors without locality constraints. Its timesharing through shared memory and paging was successful at providing both excellent interactivity and utilization [13, 12].

4.2 Policy Overview

The T3D Gang Scheduler allocates processors and barrier circuits for all programs. In order to satisfy the diverse computational requirements of our clients, the programs are classified by access requirements:

- Interactive class jobs require responsive service
- Debug class jobs require responsive service and can not be preempted
- Production class jobs require good throughput
- Benchmark class jobs can not be preempted
- Standby class jobs have low priority and are suitable for absorbing otherwise idle compute resources

There are several class-dependent scheduling parameters to achieve the desired performance characteristics.

- Priority: Job classes are prioritized for service. We make interactive jobs higher priority than production jobs during the daytime and assign them equal priority at night.
- Wait time: The maximum time that a job should wait before (or between) processor access. This is used to ensure timely responsiveness, especially for interactive and debug class jobs. After a job has waited to be loaded for the maximum wait time, an attempt will be made to reserve a block of processors for it. This processor reservation mechanism frequently preempts multiple small jobs to prevent starvation of large jobs.
- Do-not-disturb time multiplier: This parameter is multiplied by the number of processors to arrive at the do-not-disturb time, the minimum processor allocation time before preemption. A job will never be preempted before its do-not-disturb time is up. This allows the desire for timely response to be balanced against the cost of moving a job's state onto disk and back to memory (it is similar to the scheme proposed for the Tera MTA [1]). The do-not-disturb time multiplier should be set to a value substantially larger than the time required to move a job's state in one processor from memory to disk and back to memory. This time will vary with the disk configuration. On the LLNL T3D with 256 processors and 64 megabytes of memory each, the entire torus or processors can be repacked in about eight minutes or one second per processor.
- Processor limit: The maximum number of processors which can be allocated to jobs of this class. This is used to restrict the number of processors allocated to non-preemptable jobs during the daytime.

The scheduling parameters currently being used during the daytime on weekdays are shown in Table 1. The time of one year is used in several cases to insure no preemption or an indefinite wait for some job classes.

Several non-class dependent scheduling parameters also exist to regulate computer-wide resource use.

- Large job size: The minimum number of processors requested by a job for it to be considered "large". We set this to 64 during daytime.
- Large processor limit: The maximum number of processors which can be allocated to "large" jobs at any time Since "large" jobs can take a significant period of time to have their state moved between memory and disk, interactivity can be

improved by restricting the number of processors allocated to them. Our limit is 192 during daytime.

- Job processor limit: The maximum number of processors which can be allocated to any single job. We use 256, i.e. we do not place such a limit.
- System processor limit: The maximum number of processors used by jobs either running or swapped to disk. This defines the degree of overallocation; we use 576, i.e. an overallocation factor of 2.25. A limit is required to avoid filling the file system used for job state information. We are conservative in our allocation of this storage area because it is shared. Jobs will be queued, but not initiated to avoid exceeding this parameter. If an attempt is made to preempt a job when insufficient storage is available, that job will continue execution and no further attempts will be made to preempt it.

4.3 Job Scheduling Algorithm

We have implemented a two pass scheduling algorithm. The first pass checks for jobs which have waited for loading longer than their job class' maximum wait time. These jobs are viewed as having a high priority for loading and special measures are taken for loading them. If there is more than one such job, a list of these jobs is constructed then sorted by job class priority and within each priority value by the time waiting for loading. Each of these jobs is considered for loading in the sorted order. The processor requirement for the job will be compared against the scheduler's job processor limit. If the job's processor request cannot be satisfied, that job will no longer be considered a candidate for loading.

Multiple possible processor assignments for the job are considered. For each possible processor assignment, a cost is computed. The cost considers the number of nodes occupied by the potentially preempted jobs, their relative priority, and how much time remains in their do-not-disturb time. In no case will a job be preempted for another job of a lower priority class. Jobs of benchmark and debug class will never be preempted. If no possible processor assignment for loading the waiting job is located, its loading will be deferred. If a possible processor assignment is located, the lowest cost set of processors will be reserved for the exclusive use of this waiting job and jobs occupying those processors will be preempted when their do-not-disturb times have been exhausted.

Only one job will will have processors reserved for it at any point in time. Once a set of processors have been reserved for a waiting job, the reservation of processors for other waiting jobs will be deferred until the selected job has been loaded. An exception is made only in the case that a higher priority class job exceeds its maximum wait time. For example, an interactive class job could preempt the reservation of processors for a production class job. The job with reserved processors can be loaded into other processors if another compatible set of processors becomes available at an earlier time. As soon as that job is loaded, the reserved processors are made generally available. This mechanism insures timely interactivity and prevents the starvation of large jobs.

In the second scheduler pass, other executable jobs are recorded in a list sorted by job class priority and within each priority by the time waiting for loading. Each job in the sorted list is considered for processor assignment. First the limits (job processor limit, large job limit, and job class limit) are checked to determine if the job should be allocated processors. Any job satisfying these limits will have its barrier wire circuit and processor requirements considered. If the job can have its requirements met either with unallocated resources or resources which can be made available by preempting jobs which have exceeded their do-not-disturb time, it will have a barrier wire circuit and processors assigned. If a specific barrier wire is not requested, one of those available will be assigned. All four barrier wire circuits are considered for use and selected on the basis of lowest contention. More efficient relocation of jobs can be achieved by using all four barrier wire circuits.

The time required to save the state of a job on disk can be up to four minutes. Given this delay, it is not ideal to queue the loading of a job until the processors assigned to it are actually available. Whenever processors are actually made available, the job scheduler is executed again. This insures that when processors become available, they are assigned to the most appropriate jobs then available.

When a newly started job can immediately begin execution in a variety of possible sets of processor, a best-fit algorithm is used to make the selection. We also try to locate debug and benchmark class jobs, which can not be preempted, together in order to avoid blocking large jobs.

```
   h  h  c  c  a  a  a  a  CLAS JOB-USER      PID  COMMAND  #PE BASE W ST MM:SS
  h  h  c  c  a  a  a  a   Int  d - colombo   2976 icl1       8 100  1 R  42:44
 h  h  c  c  a  a  a  a    Int  h - mshaw     9529 icf3d     32 020  2 R  00:33
h  h  c  c  a  a  a  a
                          Bmrk g - grote     9264 warpslav  32 200  1 R  00:20
   h  h  c  c  a  a  a  a
  h  h  c  c  a  a  a  a   Prod a - caturla  95396 moldy    128 400  2 R 107:35
 h  h  c  c  a  a  a  a    Prod b - colombo  98057 vdif       8 000  3 R  91:54
h  h  c  c  a  a  a  a     Prod c - wenski   98484 pproto6.  32 220  3 R  85:12
                          Prod e - colombo   8712 icl3       8 004  1 R   6:23
   b  d  g  g  a  a  a  a  Prod f - colombo   8873 icl2       8 104  2 R   4:54
  b  d  g  g  a  a  a  a   Prod i - dan       5684 camille   32 020  0 O  32:08
 e  f  g  g  a  a  a  a    Prod j - vickie   99393 kiten     64 400  3 O 132:59
e  f  g  g  a  a  a  a

   b  d  g  g  a  a  a  a
  b  d  g  g  a  a  a  a
 e  f  g  g  a  a  a  a
e  f  g  g  a  a  a  a
gangster:
```

Figure 8: *Sample gangster display. The W field shows the barrier wire used. The MM:SS field shows the total execution time. The ST field shows the job's state: i = swapping in, N = new job, not yet assigned nodes or barrier wire, o = swapping out, O = swapped out, R = running, S = suspended, W = waiting job, assigned nodes and barrier wire.*

4.4 Client Interface

The default mode of operation for the Cray T3D requires all jobs, batch and interactive, to be initiated through a program called `mppexec`, which will accept as arguments the number of processors required, specific processor requirements, specific barrier wire requirements, etc. The Gang Scheduler takes advantage of this feature by creating a wrapper for `mppexec` which is upwardly compatible with it. The interface registers the job with the Gang Scheduler and waits for an assignment of processors and barrier circuit before continuing. On a heavily utilized computer, this typically takes a matter of seconds for small numbers of processors and possibly much longer for large jobs. The only additional argument to the Gang Scheduler interface is the job class, which is optional. By default, interactive jobs are assigned to the interactive job class, the Totalview debugger jobs are assigned to the debug class, and batch jobs are assigned to the production job class.

4.5 The Gangster Tool

We provide users with an interactive tool, called "gangster", for observing the state of the system and controlling some aspects of their jobs. Gangster communicates with the Gang Scheduler to determine the state of the machine's processors and individual jobs. Gangster's three-dimensional node map displays the status of each node (each node consists of two processing elements on the T3D). Gangster's job summary reports the state of each job, including jobs moving between processors and disk. Users can use gangster to change the class of their own jobs or to explicitly move their job's state to disk (suspending execution) or make it available for execution (resume).

A sample gangster display is shown in Fig. 8. This display identifies jobs in the system and assigned processors. The node map is on the left. A dot or letter denotes each node (two processing elements on the T3D): a dot indicates the node is not in use, a letter designates the job currently occupying that node. On the right is a summary of all jobs. Node number 000 is in the upper left corner of the lowest plane. The X axis extends downward within a plane. The Y axis extends up, with one Y value in each plane. The Z axis extends to the right. This orientation was selected for ease of display for a 256 processor T3D configuration.

Global limits:

Period	*User Limit*	*Run Limit*	*Aggregate mpp-pe-limit*
00:00 – 04:00	2 → 5	8 → 20	256 → 512
04:00 – 07:00	2 → 5	8 → 20	96 → 320
07:00 – 18:00	2 → 5	8 → 20	96 → 256-320*
18:00 – 24:00	2 → 5	8 → 20	192 → 320

Queue limits:

Queue Name	*User Limit*	*Run Limit*	*Job Time Limit*	*Job Processor Limit*	*Aggregate mpp-pe-limit*
pe32	1 → 3	4 → 8	4 h → 4 h	32 → 32	128 → 128
pe64	1 → 2	3 → 2-3*	4 h → 4 h	64 → 64	192 → 128-192*
pe64_long	2 → 2	2 → 2	19 h → 40 h	64 → 128	96 → 96-128*
pe128_short	1 → 1	4 → 4	15 m → 15 m	128 → 96	128 → 128
pe128	1 → 0-2*	1 → 1	4 h → 4 h	128 → 128	128 → 0-256*
pe256_short	1 → 0-1*	1 → 1	15 m → 15 m	256 → 256	256 → 0-256*
pe256	1 → 0-1*	1 → 1	4 h → 4 h	256 → 256	256 → 0-256*

*Varies by time of day and/or day of week

Table 2: *Changes made in NQS configuration parameters when the Gang Scheduler was introduced. User Limit is the maximum number of batch jobs a single user may have executing. Run Limit is the maximum number of batch jobs (from a certain queue) the system may have executing at one time. mpp-pe-limit is the maximum number of processors which the batch jobs may have allocate at one time.*

4.6 NQS Configuration

Jobs may be submitted to the Cray T3D interactively or via the NQS batch queuing system. Interactive jobs are limited to 64 processors and 2 hours. Prior to installation of the Gang Scheduler, our NQS batch system was configured to leave an adequate number of processors available for interactive computing during the day. This was changed when the Gang Scheduler was introduced, and now NQS fully subscribes the machine during the day. At night, it oversubscribes the machine by as much as 100% (Table 2).

Note that jobs requiring more than a four hour time limit were originally limited to 64 processors. Also note that substantial compute resources were sacrificed in order to insure processors for interactive computing. This was particularly noticeable in the early morning hours as the mpp-pe-limit dropped to 96 at 04:00 in order to insure the availability of 160 processors for interactive use at 08:00. Frequently this left many processors idle. Even so, under the occasionally heavy interactive workload, all processors would be allocated and interactive jobs experienced lengthy initiation delays.

With the gang scheduler, NQS is now allowed to fully subscribes the computer and oversubscribe at night. The overallocation of processors permits the execution of larger jobs and makes more jobs available for fully packing the T3D's torus of processors. NQS jobs now relinquish their processors only as needed, not in anticipation of interactive work. During periods of heavy use, this improves our realized throughput substantially while preserving good interactivity. While average interactivity has decreased slightly due to interference from NQS jobs, the worse case startup time has dropped from tens of minutes to about one minute. This is still quite acceptable to our user community, especially when accompanied by a substantial increase in realized batch throughput. We also see a few jobs relocated to better utilize the available processors during periods of heavy use, especially when jobs requiring 64 or 128 processors exist.

4.7 Performance Results

In order to quantify the effect upon system throughput and interactivity under heavy load, we have continuously tabulate system performance with the Gang Scheduler. Before installing the Gang Scheduler, we did the same for the standard UNICOS MAX scheduler and the Distributed Job Manager (DJM). DJM is a gang scheduler developed by the Minnesota

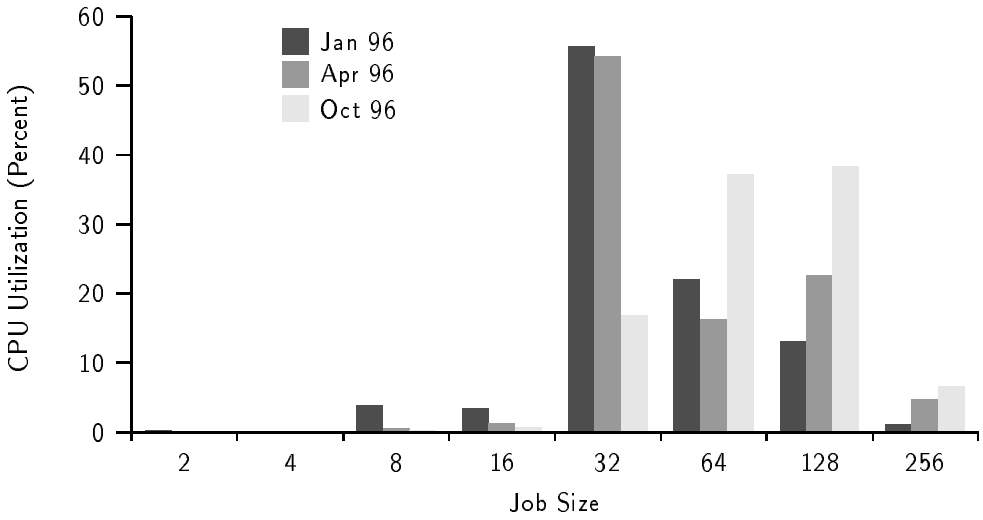

Figure 9: *Changes in workload distribution due to use of the Gang Scheduler.*

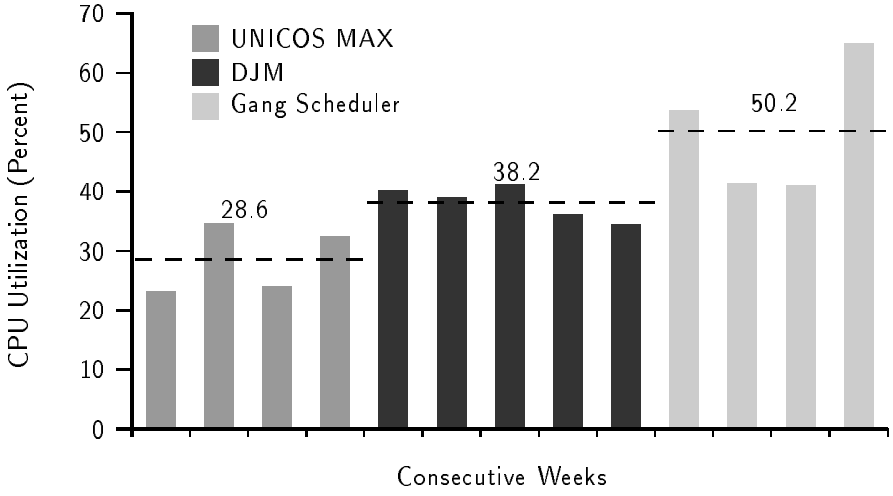

Figure 10: *Weekly utilization with the three schedulers. Dashed lines and numbers denote averages.*

Supercomputer Center, which has undergone substantial modification for performance enhancements by Cray analysts at LLNL. All of the DJM code to accomplish job swapping is new. The enhanced version of DJM was used for testing purposes.

Fig. 9 demonstrates the Gang Scheduler's ability to execute large jobs: note the dramatic improvement in throughput of 128 and 256 processor jobs. This charts the distribution of resources allocated to each job size as percentage of CPU resources actually delivered to customers. The percentage of gross CPU resources which are delivered to large jobs has increased by an even wider margin. The January 1996 period is the last full month of operation with the standard UNICOS MAX operating system. April is just after installation of the Gang Scheduler, and by October the workload had shifted to take advantage of its improved support for large jobs.

The best measure of success is probably actual throughput achieved. While utilization is quite low on weekends, the improvement in throughput at other times has dramatically improved with preemptive schedulers[2]. Fig. 10 summarizes utilization of proces-

[2]While DJM would also have provided for good interactiv-

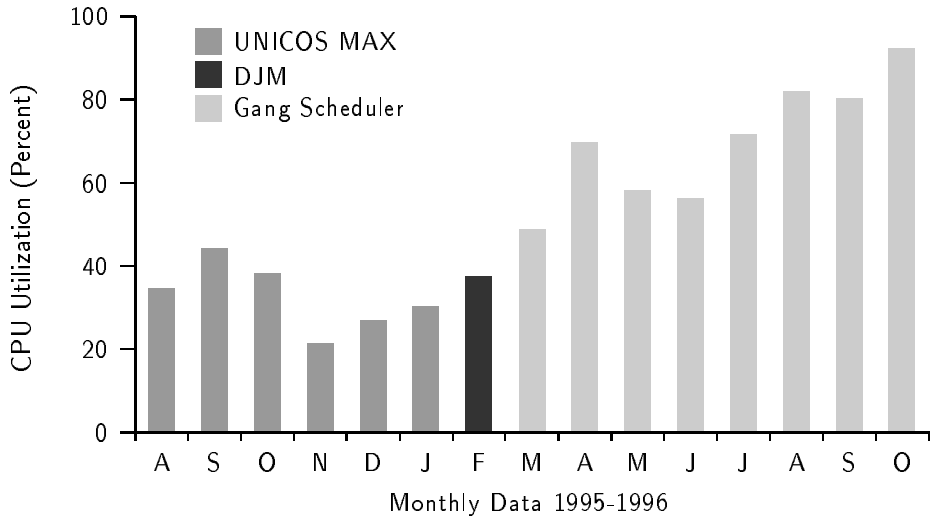

Figure 11: *Monthly utilization with the three schedulers.*

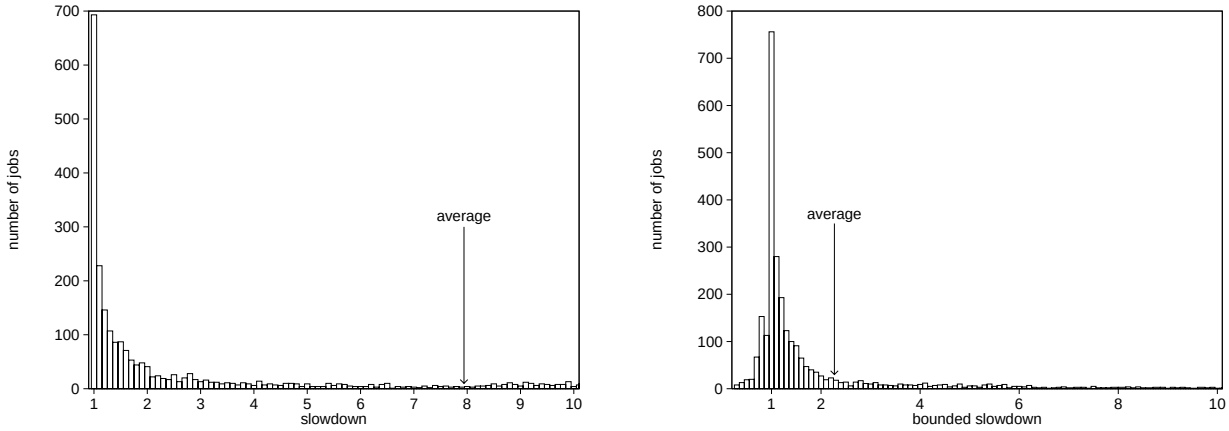

Figure 12: *Histograms of slowdown and bounded slowdown of interactive jobs.*

sor resources over the course of several entire weeks. Over the longer term, utilization has improved even more dramatically while providing good interactivity, as shown in Fig. 11. CPU utilization reported is the percentage of all CPU cycles available which are delivered to customer computation. Weekly utilization rates have reached over 96%.

Even though the improved utilization is impressive, this should not come at the expense of interactive jobs. To quantify this, we tabulated the slowdowns of interactive jobs, i.e. the ratio of the time they spent in the system to the time they actually ran. This was done for the period of July 23 through August 12. During this period, 2659 interactive jobs were run, using a total of 44.5 million node-seconds, and 1328 batch jobs were run, using a total of 289.4 million node-seconds.

The ratio of the *total* time all interactive jobs spent in the system to their *total* runtimes was 1.18, leading one to assume only 18% overhead. However, this is misleading, because slowdowns of individual jobs should be checked. The full distribution of slowdowns is plotted in Fig. 12. Actually, only the low part of

ity and throughput, it became available at the same time as our Gang Scheduler was completed and we felt that continued development of our Gang Scheduler was worthwhile. In addition, our Gang Scheduler provided the means to arbitrarily lock jobs into processors. This was important for us to be able to insure optimal throughput for jobs specified by our management.

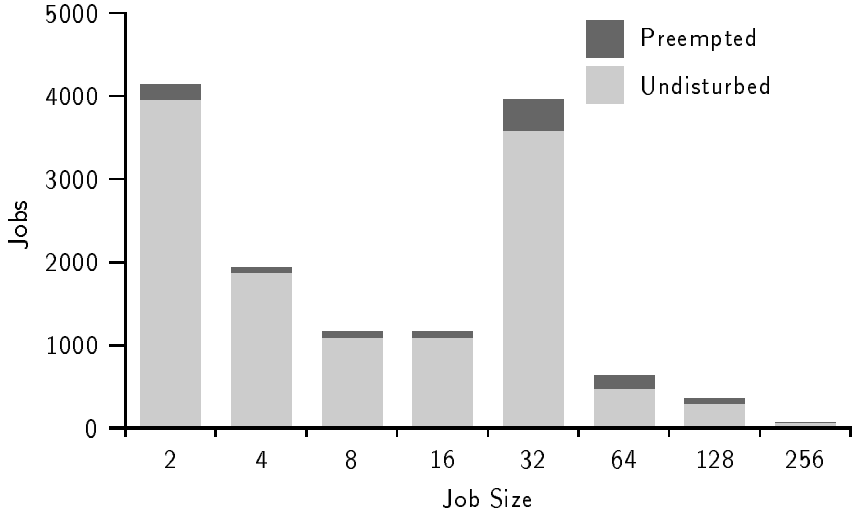

Figure 13: *Likelihood of preemption for different job sizes.*

the distribution is shown, as it has a very long tail, but most jobs have rather low slowdowns. The most extreme case was a one-second job that was delayed for just less than 23 minutes, leading to a slowdown of 1371. The average slowdown was determined to be 7.94.

While the high average slowdown is disappointing, it too is misleading. The problem is that many interactive jobs have very short runtimes, so a high slowdown may not be indicative of a real problem. Indeed, merely loading the application from disk typically takes about 0.5 seconds per processor used; thus a one second job running on 4 nodes will require two seconds of load time, for an optimal slowdown of 3, which is actually quite reasonable in this case. In order to counter the effect of short jobs, we also plot the *bounded slowdown* [11]. For long running jobs, this is the same as the slowdown. But for short jobs, the denominator is taken as the "interactivity threshold" rather than as the actual (very short) runtime. In processing the data for Fig. 12, we used a threshold of 10 seconds; the average bounded slowdown is then 2.27, and the tail of the distribution is also shorter.

4.8 Operational Characteristics

Figs. 13 to 15 portray the gang scheduler in operation. These figures are based on detailed logs of all jobs and all job preemptions during four months of production use, from June 1996 through September 1996.

It should be noted that only 7.5% of jobs that completed normally were ever preempted. Fig. 13 shows the likelihood of preemption as a function of the job size, and shows that different size jobs were preempted in roughly equal proportions. While most jobs were not preempted or were only preempted a small number of times, the maximal number of preemptions observed was pretty high: one job was preempted 77 times before completing. The histogram of number of preemptions is shown in Fig. 14.

Finally, we take a look at the average length of the time quanta for the different sizes. This is shown in Fig. 15, once for jobs that were actually preempted, and again for all jobs (i.e. including those that ran to completion without being preempted even once). As expected, use of the do-not-disturb time multiplier leads to larger average time quanta for larger jobs. (The 256-processor figure is a special case of only one job. Jobs of this size are normally not preempted.)

4.9 Future Development

While the Gang Scheduler manages the currently active jobs well, the NQS batch system selects the jobs to be started. It would be desirable to integrate the Gang Scheduler with NQS in order to more efficiently schedule all available jobs. Work is also planned for

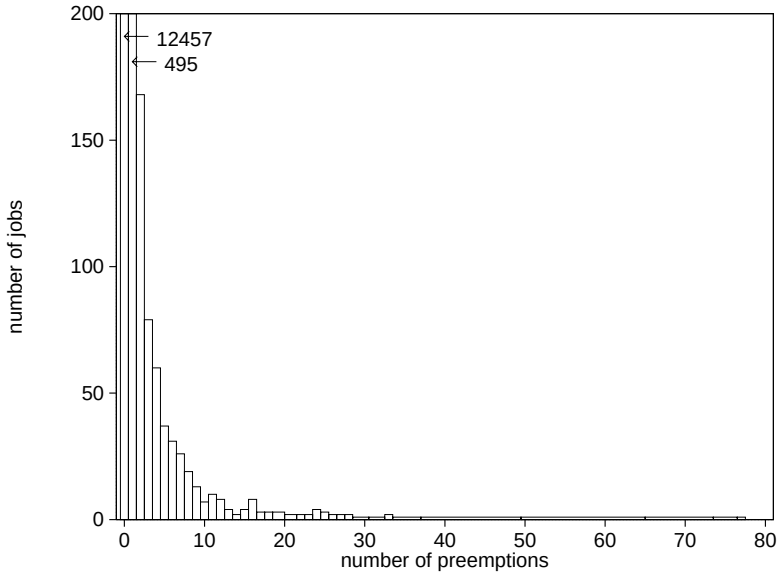

Figure 14: *Histogram of the number of preemptions suffered by different jobs.*

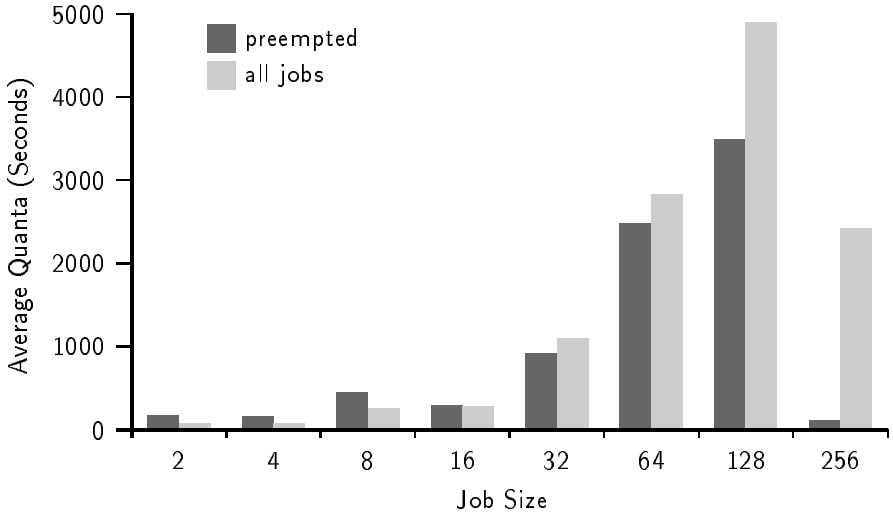

Figure 15: *Average time quanta for different size jobs.*

the gang scheduling of jobs across a heterogeneous collection of computers.

5 Conclusions

Gang scheduling has often been advocated based on its advantages of

- presenting jobs with an environment similar to that of a dedicated machine, thus allowing fine grain interactions based on user-level communication and busy waiting [9],
- allowing jobs with extreme requirements to share the system: a job that requires all the nodes does not have to wait for all previous jobs to terminate, nor does it delay subsequent jobs,
- support for interactive work by using time slicing, which guarantees a reasonable response time for short jobs, and

- not placing any restrictions or requirements on the model of computation and programming style.

However, many researchers have expressed the fear that using gang scheduling would lead to unacceptable system performance due to the overheads involved in context switching and the loss of resources to fragmentation.

In contrast, we have shown that gang scheduling can *improve* system performance significantly relative to static space slicing policies often used in practice on parallel supercomputers. Gang scheduling adds flexibility to resource allocations, and reduces the impact of bad decisions. This contributes directly to a reduction in fragmentation, and more than offsets the cost of overheads. Indeed, experience with using gang scheduling for a production workload on the Cray T3D at Lawrence Livermore National Lab has shown a dramatic increase in system utilization.

The main obstacle to widespread use of gang scheduling is memory pressure. If gang scheduling is performed at a fine granularity, all jobs need to be memory resident at the same time, so each has less memory available. The alternative is to swap jobs to disk when they are preempted, and swap them in again when scheduled. This is a viable approach, but it requires sufficient resources to be invested in adequate I/O facilities. The combination of demand paging and prefetching with gang scheduling remains an interesting topic for future research.

References

[1] G. Alverson, S. Kahan, R. Korry, C. McCann, and B. Smith, "*Scheduling on the Tera MTA*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 19–44, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.

[2] Cray Research, Inc., *Cray T3D System Architecture Overview*. Order number HR-04033, Sep 1993.

[3] D. Das Sharma and D. K. Pradhan, "*Job scheduling in mesh multicomputers*". In *Intl. Conf. Parallel Processing*, vol. II, pp. 251–258, Aug 1994.

[4] D. G. Feitelson, "*Packing schemes for gang scheduling*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 89–110, Springer-Verlag, 1996. Lecture Notes in Computer Science Vol. 1162.

[5] D. G. Feitelson, *A Survey of Scheduling in Multiprogrammed Parallel Systems*. Research Report RC 19790 (87657), IBM T. J. Watson Research Center, Oct 1994.

[6] D. G. Feitelson and B. Nitzberg, "*Job characteristics of a production parallel scientific workload on the NASA Ames iPSC/860*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 337–360, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.

[7] D. G. Feitelson and L. Rudolph, "*Distributed hierarchical control for parallel processing*". *Computer* **23(5)**, pp. 65–77, May 1990.

[8] D. G. Feitelson and L. Rudolph, "*Evaluation of design choices for gang scheduling using distributed hierarchical control*". *J. Parallel & Distributed Comput.* **35(1)**, pp. 18–34, May 1996.

[9] D. G. Feitelson and L. Rudolph, "*Gang scheduling performance benefits for fine-grain synchronization*". *J. Parallel & Distributed Comput.* **16(4)**, pp. 306–318, Dec 1992.

[10] D. G. Feitelson and L. Rudolph, "*Parallel job scheduling: issues and approaches*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 1–18, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.

[11] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong, "*Theory and practice in parallel job scheduling*". In *IPPS'97 Workshop Job Scheduling Strategies for Parallel Processing*.

[12] B. Gorda and R. Wolski, "*Time sharing massively parallel machines*". In *Intl. Conf. Parallel Processing*, vol. II, pp. 214–217, Aug 1995.

[13] B. C. Gorda and E. D. Brooks III, *Gang Scheduling a Parallel Machine*. Technical Report UCRL-JC-107020, Lawrence Livermore National Laboratory, Dec 1991.

[14] R. L. Henderson, "*Job scheduling under the portable batch system*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 279–294,

Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.

[15] S. Hotovy, "*Workload evolution on the Cornell Theory Center IBM SP2*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 27–40, Springer-Verlag, 1996. Lecture Notes in Computer Science Vol. 1162.

[16] Intel Corp., *iPSC/860 Multi-User Accounting, Control, and Scheduling Utilities Manual*. Order number 312261-002, May 1992.

[17] M. Jette, D. Storch, and E. Yim, "*Timesharing the Cray T3D*". In *Cray User Group*, pp. 247–252, Mar 1996.

[18] K. Li and K-H. Cheng, "*A two-dimensional buddy system for dynamic resource allocation in a partitionable mesh connected system*". *J. Parallel & Distributed Comput.* **12(1)**, pp. 79–83, May 1991.

[19] D. Lifka, "*The ANL/IBM SP scheduling system*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 295–303, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.

[20] C. McCann, R. Vaswani, and J. Zahorjan, "*A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors*". *ACM Trans. Comput. Syst.* **11(2)**, pp. 146–178, May 1993.

[21] C. McCann and J. Zahorjan, "*Scheduling memory constrained jobs on distributed memory parallel computers*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 208–219, May 1995.

[22] J. K. Ousterhout, "*Scheduling techniques for concurrent systems*". In 3rd *Intl. Conf. Distributed Comput. Syst.*, pp. 22–30, Oct 1982.

[23] E. W. Parsons and K. C. Sevcik, "*Multiprocessor scheduling for high-variability service time distributions*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 127–145, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.

[24] R. C. Regis, "*Multiserver queueing models of multiprocessing systems*". *IEEE Trans. Comput.* **C-22(8)**, pp. 736–745, Aug 1973.

[25] E. Rosti, E. Smirni, L. W. Dowdy, G. Serazzi, and B. M. Carlson, "*Robust partitioning schemes of multiprocessor systems*". *Performance Evaluation* **19(2-3)**, pp. 141–165, Mar 1994.

[26] E. Rosti, E. Smirni, G. Serazzi, and L. W. Dowdy, "*Analysis of non-work-conserving processor partitioning policies*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 165–181, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.

[27] B. Schnor, "*Dynamic scheduling of parallel applications*". In *Parallel Computing Technologies*, V. Malyshkin (ed.), pp. 109–116, Springer-Verlag, Sep 1995. Lecture Notes in Computer Science vol. 964.

[28] K. C. Sevcik, "*Application scheduling and processor allocation in multiprogrammed parallel processing systems*". *Performance Evaluation* **19(2-3)**, pp. 107–140, Mar 1994.

[29] K. C. Sevcik, "*Characterization of parallelism in applications and their use in scheduling*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 171–180, May 1989.

[30] A. Tucker and A. Gupta, "*Process control and scheduling issues for multiprogrammed shared-memory multiprocessors*". In 12th *Symp. Operating Systems Principles*, pp. 159–166, Dec 1989.

[31] M. Wan, R. Moore, G. Kremenek, and K. Steube, "*A batch scheduler for the Intel Paragon with a non-contiguous node allocation algorithm*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 48–64, Springer-Verlag, 1996. Lecture Notes in Computer Science Vol. 1162.

[32] K. Windisch, V. Lo, R. Moore, D. Feitelson, and B. Nitzberg, "*A comparison of workload traces from two production parallel machines*". In 6th *Symp. Frontiers Massively Parallel Comput.*, pp. 319–326, Oct 1996.

[33] Q. Yang and H. Wang, "*A new graph approach to minimizing processor fragmentation in hypercube multiprocessors*". *IEEE Trans. Parallel & Distributed Syst.* **4(10)**, pp. 1165–1171, Oct 1993.